# C++QED: An object-oriented framework for wave-function simulations of cavity QED systems

A. Vukics* and H. Ritsch

*Institute of Theoretical Physics, University of Innsbruck,*
*Technikerstrasse 25, A-6020 Innsbruck, Austria*

We present a framework for efficiently performing Monte Carlo wave-function simulations in cavity QED with moving particles. It relies heavily on the object-oriented programming paradigm as realised in C++, and is extensible and applicable for simulating open interacting quantum dynamics in general. The user is provided with a number of "elements", eg pumped moving particles, pumped lossy cavity modes, and various interactions to compose complex interacting systems, which contain several particles moving in electromagnetic fields of various configurations, and perform wave-function simulations on such systems. A number of tools are provided to facilitate the implementation of new elements.



## I. INTRODUCTION

Based on our experience gained in recent years in Monte Carlo wave-function (MCWF) simulations of simple moving-particle cavity QED (CQED) systems performed with low-level codes [1, 2, 3, 4], we have decided to summarise our know-how on the problem by developing a high-level framework for such simulations. The framework is highly modular and therefore easy to maintain, relies solely on standard C++ programming techniques and therefore portable, and provides an interface which is easy to use even for those not so familiar with the theoretical models of moving-particle CQED ([5] is a review of the theory involved). Meanwhile, thanks to the optimisation mechanisms of C++ compilers, we are safe to claim not to have noticeably lost in efficiency as compared to our previous low-level codes. Potentially, the framework is of good use for the quantum optics community.

Simulating moving quantum particles presents many non-trivial numerical problems especially of stability [1, 4]. Hence, in the framework very careful numerics is needed. Accordingly, as discussed in App. A, we use a slightly modified version of the original MCWF algorithm (cf eg [6]) involving the use of adaptive step-size ODE steppers and interaction picture.

At present the framework consists of three parts: The first part is the MCWF driver (Sec. III),

* andras.vukics@uibk.ac.at

which has only an abstract view on the open system to be simulated, represented by an abstract class. This abstract class stands at the origin of a class hierarchy consisting the second part of the framework (Sec. IV). Eg a system can be an element system or a composite system containing several element systems. The aim of the hierarchy is to provide the user with tools to build composite systems from several elements, and to facilitate the implementation of such elements. Clearly, the first two parts stand quite independently of each other and are also generally applicable. As the third part of the framework several elements are provided at the lower levels of the hierarchy intended as building blocks for systems of moving-particle CQED (Sec. V). The building blocks are pumped moving particles, pumped lossy cavity modes, pumped two-level atoms, and interactions between them eg interaction between a cavity mode and a pumped particle moving along or orthogonal to it. This third part is independent of the first part, but not, of course, of the second part, the elements stemming from the same class hierarchy.

For a given system on the highest level the user is required to write a simple driver program in C++ in which he/she defines the system to be simulated using the elements (selecting a number of free elements and interactions between them) and passes this system to the MCWF driver, which then evolves the system on a number of Monte Carlo trajectories. A description of the user interface and example drivers are given in Sec. II.

Note that our approach here is quite different from the one presented in [7]. That approach is built on a hierarchy of classes representing Hilbert space operators and state vectors, and the application of operators on vectors is defined. Operators acting on complex systems can then be built from elementary operators using direct product. A similar idea is implemented in the popular Quantum Optics Toolbox for MATLAB [8]. Consider for a moment how this approach could be applied for moving particles: In this case dealing with both operators $x$ and $p$ cannot be avoided. A moving-particle state vector can be stored in either representation (the state-vector object stores in which representation it is at the moment), and when the other operator is to be applied, an in-place Fast Fourier Transformation (FFT) is needed. However, as our experience shows, such a transformation always has numerical errors, which can disturb careful statistics.

In our approach the user is provided with a much higher level interface, our classes representing whole physical systems instead of Hilbert space operators. This is certainly at the cost of flexibility, but our framework does not aim at such generality as the above, since it has been developed with a more concrete problem in mind, in particular, CQED with moving particles. For this given problem we consider our approach as more efficient than the above, since, as we will show in Sec. V we can completely avoid in-place FFT.

In the following we first present the highest level of the framework, that is, the user interface, so that the reader can immediately get a feeling about our approach. Also, by reading Sec. II the reader can in principle already use the framework, so this can be considered as a short write-up. This is followed by the long write-up, the presentation of the different parts of the framework. We include sections entitled "Desideratum" in which we indicate features that would logically belong to the given part, but are as yet missing because we have not yet needed them. These may easily be implemented in the future.

Finally, in Sec. VI, we summarise our test runs performed with the framework. In the Appendices we describe our version of the MCWF method and the most important modules used in the framework.

The source code contains more than 60 source files and a totality of about 4000 lines, and is distributed in `tgz` format. It can be get either from SourceForge.net at http://sourceforge.net/projects/cppqed/ or directly from the authors. The framework has been tested under Debian GNU/Linux and RedHat Linux operating systems, in both cases the GNU C++ compiler has been used for compilation.

## II. THE USER INTERFACE

### A. Writing drivers

The classes a user has to know about are listed in Tab. I together with the most important parameters, which will be explained further down in the text. The set of elements for systems in moving-particle CQED is explained in detail in Sec. V.

To ease the understanding of the framework's workings example drivers are given in Figs. 1 and 2. The driver in Fig. 1 simulates one single particle moving in a ring cavity, that is, two travelling-wave modes propagating in opposite directions. Both modes are lossy and one of them is pumped. In addition, the particle can also be pumped and scatter light from the pump into the modes. The driver in Fig. 2 describes two identical particles moving orthogonal to the axis of a single-mode cavity in a standing-wave pump field.

The user has to choose an appropriate set of free systems and the interactions between them, and instantiate the corresponding `Free` and `Interaction` classes with the appropriate parameters. If two elements are exactly identical, only one object is needed. This is the case eg with several identical particles: one instant of the `MovingParticle` class stands for all of them (an example of

| | |
|---|---|
| **Elements** | |
| **Frees** | |
| `LossyMode` | $\Delta_{\mathrm{C}}$, $\kappa$, `photonCutoff` |
| `PumpedLossyMode` | ", $\eta$ |
| `MovingParticle` | $\omega_{\mathrm{recoil}}$, `momentumCutoff` |
| `PumpedMovingParticle` | ", $\eta_{\mathrm{eff}}$, $K_{\mathrm{pump}}$, `pumpModeFunction` |
| **Interactions** | |
| `ParticleOrthogonalToCavity` | `cavity`, `pumpedParticle`, $U_0$ |
| `ParticleAlongCavity` | `cavity`, `(pumped)particle`, $U_0$, $\eta_{\mathrm{eff}}$, $K_{\mathrm{cavity}}$, `cavityModeFunction` |
| `ParticleCavity2D` | `cavity`, `particle`, `pumpedParticle`, $U_0$, $\eta_{\mathrm{eff}}$, $K_{\mathrm{cavity}}$, `cavityModeFunction` |
| `ParticleTwoModes` | `particleCavity1`, `particleCavity2` |
| `IdenticalParticles` | `(pumped)particle`, $N_{\mathrm{particle}}$, `vector<`$\|\phi_{\mathrm{particle}}\rangle$`>` |
| **Composite** | `vector<SubsystemsInteraction>` |
| **SubsystemsInteraction** | `Interaction&`, `vector<subsystemSequentialNumber>` |
| **HS_Vector** | `dimension` |
| **Trajectory** | $\|\Psi(t=0)\rangle$, `OpenSystem&`, `seed`, `eps`, `dplimit` |

Table I: Classes constituting the user interface of the framework, with the set of elements extendable in the future at will. Next to each class their most important parameters are listed, these are explained in the text. The `Interaction`s take references to their subsystems as parameters — `cavity` is an instant of class `(Pumped)LossyMode`, `(pumped)particle` one of `(Pumped)MovingParticle`, while `particleCavity` one of `MovingParticleCavity` cf Sec. V C 1.

this can be seen in Fig. 2).

The `Free` objects are then to be (virtually) arranged into a sequence starting with number 0, and the user has to create a `vector` of `SubsystemsInteraction` class objects. The latter is a helper class for the `Composite` class, storing a reference to an `Interaction` and the sequential number of those `Free` objects between which the given interaction acts. Most interactions will be between two subsystems, but we have found cases with interactions between three or four subsystems (cf Sec. V). The `IdenticalParticles` class is an `Interaction` between all the particles, that is, an arbitrary number of subsystems in principle.

When giving the sequential numbers the user has to remain consistent with the originally conceived sequence of the `Free` objects, and the order of the subsystems in an `Interaction` object is also important. Eg in Fig. 1 Line 19 instead of `(pc1,1,0)` it would be an error to write `(pc1,1,2)` because `Free` Nr. 2 is a `PumpedLossyMode` and not a `MovingParticle`, but also `(pc1,0,1)` because `ParticleAlongCavity` is an interaction between a `LossyMode` and a `MovingParticle` and not vice

versa. Such errors cause an exception during the construction of the `Composite` object.

The free systems provide helper functions to prepare state vectors (of class `HS_Vector`) characteristic to the given system. Eg for a `LossyMode` object one can prepare a Fock state or a coherent state. This, together with the possibility of making up direct products of several state vectors, facilitates the preparation of initial conditions. Eg in Fig. 1 Line 26 we prepare a state in which the particle has a wave packet centred at position `x0` with momentum `k0` and spread `xsig`, the $+K$ mode is in a coherent state with complex amplitude `alpha`, and the $-K$ with `beta`. Here again, we have to comply with the our preconceived order of the `Free` objects in the sequence.

As output such a program first summarises the parameters of the system, then at certain time instants (whose frequency is specified by the user) displays the time and the time step followed by a set of quantum averages specified in the element system classes. At specified time instants the whole state vector is displayed, but in practice this can be too big to store and gain information from. An example output is given in Fig. 3.

### B. Desideratum

With some effort the preparation of drivers could be made automatic, such that the user is presented with a higher level interface in which he/she specifies the system using some simple formal language, and then the framework writes and compiles the C++ driver corresponding to the system. A similar idea can be found implemented in the XMDS package [9].

## III. EVOLUTION

### A. MCWF trajectories

What we expect from a MCWF trajectory driver class (called `Trajectory` in our framework); what parameters does it need and what functionalities should it provide?

First we need to represent the state vector of the system. The most straightforward representation is a *complex packed array* (CPA), that is, a real array, in which the real and imaginary parts of the state-vector amplitudes are placed in alternate neighbouring elements. In our framework the low-level notion of a CPA is furnished with an interface class called `HS_Vector` (for Hilbert-space vector) supplying the operations we expect for a vector of a Hilbert space. These include algebraic (vector-space) operations including direct product of several vector spaces, metric operations, and both low and high level access to amplitudes. When instantiating a `Trajectory` the initial condi-

tion of the system has to be given in an appropriate instant of `HS_Vector` and this is eventually *replaced* by the driver when evolving the system.

Every system must supply an interface towards the trajectory driver containing the operations needed to perform a MCWF step on the system as described in App. A. This interface is the abstract view the driver has on the system to be simulated. In the present framework such an abstract system is represented by an abstract class called `OpenSystem`. The hierarchical implementation of this interface for more and more concrete systems constitutes the main part of the work presented here and is described in Sec. IV. In the C++ implementation of the object-oriented paradigm, an abstract class cannot be instantiated but can be referred to by a reference (a pointer), to preserve run-time polymorphism. Hence, a `Trajectory` object takes a reference to an `OpenSystem`.

An ODE integrator and a random-number generator are needed to perform Step 1 and 2 of an MCWF step, respectively. These are also wrapped into interface classes called `Evolved` and `Randomized`, respectively. At the moment, these classes are implemented using the Gnu Scientific Library (GSL) [10], but here a user is free to choose his/her own favourite library (eg Numerical Recipes) or even hand-crafted code. To better localise object creation, only "factory" objects for these classes are passed to the `Trajectory` object (for a description of the factory-class and other programming techniques appearing in this paper see [11]).

Other important parameters are the highest allowed jump probability `dplimit` and the relative precision for the ODE stepper `eps`.

The class supplies a member function called `Step` to perform one adaptive-stepsize MCWF step on the system as follows:

1. Invokes the ODE stepper to evolve the state vector according to Eq. (A2) for a suitable time interval `dtdid`. $H_{\mathrm{nH}}$ for the system is taken from the `OpenSystem` class.

2. Performs the additional (exact) part of the evolution as $|\Psi(t+\delta t)\rangle = U^{-1}(\delta t)\,|\Psi_{\mathrm{I}}(t+\delta t)\rangle$.

3. Examines whether a jump should be made. For this it uses a random number, `dtdid`, and a system-specific jump function again taken from the `OpenSystem` class.

4. The ODE stepper supplies a time step `dttry` which is likely to work for the *next* step. The driver examines whether the jump probability would have overshoot `dplimit` were it calculated with `dttry` instead of `dtdid`. If this is the case, `dttry` is reduced.

5. Calculates and communicates towards the user physical properties of the system at the given time instant, such as the state vector itself and/or important quantum averages — exactly

what is again taken from `OpenSystem`.

A number of helper functions are provided to take not only a step but evolve a whole trajectory or an ensemble average of trajectories.

### B. Desideratum

Other methods of wave-function simulation of open systems can be straightforwardly added to the framework, although the `OpenSystem` interface may need to be extended. These include the quantum state diffusion method [12], and the orthogonal quantum jump method [13]. It would be advisable to keep a common interface for the different drivers, so that the same helper functions work for all of them.

Wave function simulations can very efficiently be done parallel. With additional helper functions parallel execution can be easily implemented.

## IV. SYSTEM HIERARCHY

Every class derived from `OpenSystem` is an `OpenSystem`, features the same interface, and hence can be passed to the trajectory driver.

As indicated in Fig. 4 an `OpenSystem` is either `Composite` or `Element` system. Element systems can be used as building blocks to compose composite systems. One may wonder why derive also `Element` from `OpenSystem` when elements are simple systems with known behaviour, so that one is unlikely to wish to simulate such systems. The answer is that one *may* wish to simulate them for testing purposes when implementing a new `Element` class. Also, this way quite an amount of code can be reused.

An `Element`, in turn, can be either a `Free` system or an `Interaction` of such systems. We emphasise the fact that an `Interaction` is also an `Element`, and hence an `OpenSystem`. One is even less likely to wish to simulate only the interaction part of the dynamics without the free systems: The reason for this arrangement is again code reuse.

We note that we had considered the alternative design depicted in Fig. 5. Here, there is a very clear distinction between system that use interaction picture and those that do not. In many sense this design is more logical and attractive, since it grasps better the structure of the problem. However, it involves the use virtual bases, consisting a slight efficiency overhead, and,

more importantly, a bigger overhead in the complexity of the code. We therefore eventually resorted to the first simpler design for the testing phase.

The design we have found ultimately useful is, however, the one depicted in Fig. 6. This one unites the advantages of the previous two, without the overhead of virtual bases. This design is uncompromising in the sense that it is very clearly expressed which virtual functions a class at the lower levels of the hierarchy has to implement.

Although the underlying design in our framework is this last one, in the following, for the sake of simplicity, and to ease the understanding for those not so familiar with object-oriented programming, we go on presenting the framework as if the underlying design was the *first* one. The differences are purely technical throughout.

### A. `OpenSystem`

The `OpenSystem` class is not a purely abstract one, since it has one data member: the dimension of the system — a parameter every quantum system has in common. In addition it features a number of virtual functions (function prototypes) which enable the driver class to perform a MCWF step as described in Sec. III A. Eg the (non-Hermitian) Hamiltonian of the system is implemented by the function

```
void H ( double t, const double* Psi, double* dPsidt, const CPA_View& V );
```

The first three arguments are the expected ones: time, an array for the state vector $|\Psi\rangle$, and one for the state-vector derivative $d|\Psi\rangle/dt$. It is the last parameter that needs some explanation. Since an `OpenSystem` can be an `Element` system, it must be prepared to be embedded into a complex system as a subsystem. If so, to be able to perform the operation on the state vector of the whole system, `H` must have some information about the embedding complex system. As explained in App. B, this information can be condensed into a set of *array slice*s, which set, in turn, is implemented by a class called `CPA_View` in our framework.

The other important virtual member functions are `U`, `J`, and `Display`, which take care of Phases 2, 3, and 5 of a MCWF step as described in Sec. III A, respectively. They all take arguments one would expect them to, *plus* a `CPA_View`.

A further important virtual member function is called `HighestFrequency`, and returns the highest characteristic frequency in the system's time evolution — a measure what every dynamical system is expected to have. This is needed by the `Trajectory` driver to initiate the ODE stepper:

adaptive step-size ODE steppers need a good guess for the initial time step to try, which is derived by the driver from the highest characteristic frequency of the system.

## B. `Element`

At the level of `OpenSystem` the functions `H`, `U`, `J`, and `Display` are *virtual* functions because we can not tell what these functions are to do for a general `OpenSystem`.

An `Element` system will be mostly embedded into a complex system as a subsystem. As explained in detail in App. B to calculate eg the Hamiltonian it has to iterate over the state-vector slices contained by its `CPA_View`, which corresponds to iterate over all the possible combinations of the quantum numbers of other subsystems — the "dummy" quantum numbers from the given subsystem's point of view, and call the same function on the corresponding slice. Function `H` is implemented accordingly, and class `Element` hence features the virtual function

```
void H_elem ( const double* Psi, double* dPsidt, const CPA_Slice& S ) const;
```

Note that the time argument is not passed over to `H_elem`. The time dependence of the original Hamiltonian `H` is rather taken care of by another virtual function `H_update`, which updates the inner state of the object if it does not correspond to the given time instant. With this method much calculation can be saved when the same object is used to describe several identical subsystems.

Note that `Element` is also an abstract class because although it implements function `H` from `OpenSystem`, it declares new virtual functions, which must be implemented further down in the hierarchy.

`J` and `Display` are implemented along similar lines as `H`, in both cases new virtual functions are declared. Eg for `J` we need a function `J_dpoverdt` which calculates the probability of a jump per unit time in the given subsystem, and a function `J_elem` which actually performs the jump on a given state-vector slice if required.

`U` is not implemented by `Element`. An `Element` can be `Free` or `Interaction`. `U` represents the part of the dynamics which can be exactly solved, that is, the part of the Hamiltonian which can be diagonalised. This is possible for some free systems, but not for interactions. Therefore `U` is implemented only in class `Free`, along exactly the same lines as `H` in class `Element`.

Interactions may affect the parameters of frees. A straightforward example for this is a cavity mode whose resonance frequency is shifted when interacting with an atom. Hence, class `Interaction` features a virtual function called `FreesAdjust`, which performs the required mod-

ification in the parameters of the free systems. It is important to note that this is done at the construction of `Composite` rather than at the construction if the given `Interaction`. Indeed, at the construction of the interaction we do not yet know how many times it will be applied: this becomes clear only when we already know the layout of the whole composite system — in the above example the cavity frequency has to be shifted *twice* if there are two atoms instead of one.

Not every element has to implement all the virtual functions declared in class `Element`. Eg we can easily imagine free systems whose dynamics can be exactly solved. In this case the coherent evolution is completely taken care of by `U`, hence `H_elem` and `H_update` need not be implemented. An other common case is when an element's dynamics is purely coherent. In this case the functions connected to `J` are not implemented. An interesting case is that of `IdenticalParticles` cf Sec. V C 4, which can be considered the extreme: this class exists solely to perform calculations in occupation-number representation, and implements solely the functions related to `Display`.

### C. `Composite`

A very important task of class `Composite` is to keep track of its elements (frees and interactions) and their `CPA_View`s. The calculation of the `CPA_View`s takes place already at the construction of the `Composite` object.

`Composite` is a concrete type, so that it has to implement all the virtual functions of its parent class `OpenSystem`. Eg `H` is implemented as calling successively the `H` of each element with the `CPA_View` of the given element. For this to work, it is important that the `H` functions of the elements *add* their contribution to `dPsidt` rather than replace it. Hence with the successive calls the contributions of elements add up, according to the model (B1).

The implementation of the composite `U` and `Display` is rather similar, only `J` needs a bit more elaboration, since here the element `J`s should not be performed one after the other, but a *choice* has to be made as to which one (if any) to perform. The interested reader should refer to the code to see how this is implemented.

### D. Desideratum

It is an interesting possibility, and one whose implementation should not be too difficult in the framework to allow composite systems to be elements of even more composite systems. This would be useful eg to facilitate the simulation of several atoms of complex structure.

## V. EXAMPLE: MOVING PARTICLES IN CAVITY

### A. Theory

Let us consider a single pumped two-level atom interacting with a single pumped lossy cavity mode. Our units are chosen such that $\hbar = 1$. Using the Jaynes-Cummings model to describe the arising interactions, the Hamiltonian for such a system reads ($a$ is the cavity field operator, the $\sigma$s are the atomic internal operators, $\mathbf{r}$ and $\mathbf{p}$ are the atomic position and momentum operators)

$$H = -\Delta_{\mathrm{C}}\, a^\dagger a + i\left(\eta a^\dagger - \eta^* a\right) + \frac{\mathbf{p}^2}{2\mu} - \Delta_{\mathrm{A}}\,\sigma_z + i\left(\eta_{\mathrm{t}}^*(\mathbf{r})\sigma - \eta_{\mathrm{t}}(\mathbf{r})\sigma^\dagger\right) - i\left(g(\mathbf{r})\sigma^\dagger a - g^*(\mathbf{r})a^\dagger\sigma\right), \quad \text{(1a)}$$

where the terms describe free field, pumping of the mode, atomic external and internal degrees of freedom (free and pumped), and atom-mode interaction, respectively. The Liouvillean reads

$$\mathcal{L}\rho = \kappa\left(2a\rho a^\dagger - \left[a^\dagger a, \rho\right]_+\right) + \gamma\left(2\int d^2\mathbf{u}\, N(\mathbf{u})\,\sigma e^{-ik_{\mathrm{A}}\mathbf{ur}}\rho\, e^{ik_{\mathrm{A}}\mathbf{ur}}\sigma^\dagger - \left[\sigma^\dagger\sigma, \rho\right]_+\right), \quad \text{(1b)}$$

where the first term describes cavity decay and the second one atomic spontaneous emission. The second term contains momentum recoil due to spontaneous emissions. The unit vector $\mathbf{u}$ is the direction of the spontaneously emitted photon, and $N(\mathbf{u})$ the direction distribution characteristic to the given atomic transition.

The operator (1b) conforming with Eq. (A1) we can immediately read the necessary jump operators for this system. There is one for cavity decay and an infinite set parametrised by $\mathbf{u}$ for atomic decay:

$$J_{\mathrm{C}} = \sqrt{2\kappa}\, a, \quad J_{\mathrm{A}}(\mathbf{u}) = \sqrt{2\gamma}\, e^{-iK\mathbf{ur}}\,\sigma. \quad (2)$$

We introduce $Z_{\mathrm{C}} = \kappa - i\Delta_{\mathrm{C}}$, $Z_{\mathrm{A}} = \gamma - i\Delta_{\mathrm{A}}$. The non-Hermitian Hamiltonian is obtained by replacing $\Delta_{\mathrm{C}}$ with $iZ_{\mathrm{C}}$ and $\Delta_{\mathrm{A}}$ with $iZ_{\mathrm{A}}$ in Eq. (1a).

In the limit of large atomic detuning $\Delta_{\mathrm{A}}$ the atomic internal degree of freedom $\sigma$ can be adiabatically eliminated, as described in Refs. [2, 5]:

$$\sigma \approx \frac{g(\mathbf{r})\, a + \eta_{\mathrm{t}}(\mathbf{r})}{i\Delta_{\mathrm{A}} - \gamma}. \quad (3)$$

In this limit the atomic spontaneous emission can be neglected in most cases of interest. We will resort to this approximation to simplify the discussion. Putting $\gamma = 0$ leaves us with only one jump operator

$$J_{\mathrm{C}} = \sqrt{2\kappa}\, a. \quad \text{(4a)}$$

| Free | $U(t)$ | $H$ | J_dpoverdt | J_elem | Display |
|---|---|---|---|---|---|
| LossyMode | $\exp\left(-Z_{\mathrm{C}} t\, N\right)$ | $\varnothing$ | $2\kappa\, N$ | a | $\langle N\rangle$, $(\Delta N)^2$, $\langle a\rangle$ |
| PumpedLossyMode | $\Uparrow$ | $i\left(\eta a^\dagger - \eta^* a\right)$ | $\Uparrow$ | $\Uparrow$ | $\Uparrow$ |
| MovingParticle | $\exp\left(-i\omega_{\mathrm{rec}} t\, k^2\right)$ | $\varnothing$ | $\varnothing$ | $\varnothing$ | $\langle k\rangle$, $(\Delta k)^2$, $\langle x\rangle$, $\Delta x$ |
| PumpedMovingParticle | $\Uparrow$ | $\eta_{\mathrm{eff}} \lvert m(\xi)\rvert^2$ | $\Uparrow$ | $\Uparrow$ | $\Uparrow$ |

Table II: Summary of the free elements' functionality, fully exposed in the text. $\Uparrow$ indicates that the given function is inherited from the parent class. $N = a^\dagger a$ is the photon number of the mode.

We plug (3) into (1a). We take $g(\mathbf{r}) = g f(\mathbf{r})$ and $\eta_{\mathrm{t}}(\mathbf{r}) = \eta_{\mathrm{t}}\zeta(\mathbf{r})$, and assume that $g$ and $\eta_{\mathrm{t}}$ are real (the possibility of their being complex is investigated in [14]). We obtain the following effective non-Hermitian Hamiltonian

$$H_{\mathrm{eff}} = -\left(iZ_{\mathrm{C}} - U_0\left|f(\mathbf{r})\right|^2\right) a^\dagger a + i\left(\eta a^\dagger - \eta^* a\right) + \frac{\mathbf{p}^2}{2\mu} + \eta_{\mathrm{eff}}\left|\zeta(\mathbf{r})\right|^2 + \mathrm{sign}(U_0)\sqrt{U_0\,\eta_{\mathrm{eff}}}\left(f^*(\mathbf{r})\,\zeta(\mathbf{r})\,a^\dagger + \mathrm{h.c.}\right), \quad (4b)$$

with $U_0 = \left|g\right|^2/\Delta_{\mathrm{A}}$, $\eta_{\mathrm{eff}} = \left|\eta_{\mathrm{t}}\right|^2/\Delta_{\mathrm{A}}$.

The following set of elements realizes the system (4). An important restriction whose reason will become apparent later in this section is that the mode functions are restricted to one dimension and either standing- or travelling-wave modes:

$$f(\mathbf{r}),\, \zeta(\mathbf{r}) = m(\xi) \equiv \begin{cases} \sin(K\xi) \\ \cos(K\xi)\,, \\ e^{\pm iK\xi} \end{cases} \quad \xi = x, y, z. \quad (5)$$

### B. Free elements

These classes implement H_elem, H_update, J_dpoverdt, J_elem, and the functions connected to Display: Average and AverageProc from parent class Element, and U_elem, U_update from parent class Free. Their functionality is summarised in Tab. II.

#### 1. *(Pumped)LossyMode*

Class LossyMode implements the dynamics of a free lossy (cavity) mode. Its parameters are the detuning between the driving and the cavity resonance $\Delta_{\mathrm{C}}$, the cavity decay rate $\kappa$, and the photon number cutoff.

The non-Hermitian Hamiltonian can be diagonalised exactly, so that `H_elem` needs not be implemented while `U_elem` is implemented as applying $U(t) = \exp\left(-Z_{\rm C} t\, a^\dagger a\right)$ on the state vector slice.

A `PumpedLossyMode` has the additional parameter $\eta$. Here only `H_update` and `H_elem` needs to be implemented to apply the Hamiltonian

$$H_{\rm I}(t) = iU^{-1}(t)\left(\eta a^\dagger - \eta^* a\right)U(t) = i\left(\eta e^{Z_{\rm C}t} a^\dagger - \eta^* e^{-Z_{\rm C}t} a\right). \tag{6}$$

Since pumping does not affect the remaining part of the dynamics, all the other functions are exactly the same as for `LossyMode`, and `PumpedLossyMode` indeed has access to these functions: "inherits" them from the parent class `LossyMode`. This is the reason why in the class inheritance hierarchy in Fig. 4 `PumpedLossyMode` stems from `LossyMode`. Clearly, this technique can be applied to reuse a lot of code, and has indeed been applied throughout in our framework.

### *2. (Pumped)MovingParticle*

A similar relationship exists between `MovingParticle` and `PumpedMovingParticle`. `MovingParticle` implements the dynamics of a free quantum mechanical particle moving in 1D, with Hamiltonian $H = p^2/(2\mu)$. This Hamiltonian is most conveniently implemented in momentum basis. For the numerics the momentum basis must be discrete, which amounts to some finite quantisation volume (length). Our choice of units is such that the smallest momentum is $\Delta k = 1$, that is, the quantisation length is $2\pi$. It is easy to see that the use of discrete momentum basis entails periodic boundary condition at the borders of the quantisation length. The parameters are the recoil frequency $\omega_{\rm rec} \equiv \hbar\,\Delta k^2/(2\mu) = 1/(2\mu)$ and the spatial resolution. The latter has to be an integer power of 2 to be able to perform radix-2 FFT on the state vector. With our units $H = \omega_{\rm rec} k^2$ with operator $k \equiv p/(\hbar\,\Delta k) = p$.

The Hamiltonian is diagonal in the momentum basis, and is quadratic in the momentum. According to our experience, the second property makes it essential to use interaction picture because the quadratic growth of the frequency is too quick for the stepper routine and results in instabilities for the higher momentum components.

According to this discussion class `MovingParticle` implements $U(t) = \exp\left(-i\omega_{\rm rec} t\, k^2\right)$, which is diagonal in momentum basis. The quantum averages calculated and communicated towards the user are: $\langle k\rangle$, $\left\langle k^2\right\rangle - \langle k\rangle^2$ (proportional to the kinetic temperature of the particle), $\langle x\rangle$, $\sqrt{\langle x^2\rangle - \langle x\rangle^2}$. This means that at each call of `Display` for the class, a Fourier transformation

| `Interaction` | $H$ | Display |
|---|---|---|
| `ParticleOrthogonalToCavity` | $A\left(m(\xi)\,a^\dagger + \text{h.c.}\right)$ | $\varnothing$ |
| `ParticleAlongCavity` | $U_0\,\lvert m(\xi)\rvert^2\,N$ + " | $\varnothing$ |
| `ParticleCavity2D` | $U_0\,\lvert m_1(\xi_1)\rvert^2\,N + A\left(m_1^*(\xi_1)\,m_2(\xi_2)\,a^\dagger + \text{h.c.}\right)$ | $\varnothing$ |
| `ParticleTwoModes` | $\sqrt{U_{01}U_{02}}\left(m_1^*(\xi_1)\,m_2(\xi_2)\,a_1^\dagger\,a_2 + \text{h.c.}\right)$ | $\varnothing$ |
| `IdenticalParticles` | $\varnothing$ | $\langle 2,0 \mid \Psi\rangle$, $\langle 1,1 \mid \Psi\rangle$, $\langle 0,2 \mid \Psi\rangle$ |

Table III: Summary of the interaction elements' functionality. $N = a^\dagger a$ is again the photon number, $A = \text{sign}(U_0)\sqrt{U_0\,\eta_{\text{eff}}}$.

has to be performed on a copy of the state vector to calculate the averages of operator $x$. This is done using the radix-2 FFT routine supplied by GSL, but here again the user is free to use his/her own favourite routine. We emphasise, however, that the time evolution is performed purely in momentum representation, nor is our `Trajectory` driver prepared to perform FFT during evolution. When FFT is performed at all, it is on a *copy* of the state vector, not an in-place transformation. Hence, we avoid numerical errors accumulating in the state vector, and also save the inverse transformation (although we lose time by copying).

`PumpedMovingParticle` implements the Hamiltonian $H_{\text{I}}(t) = \eta_{\text{eff}}\,U^{-1}(t)\,\lvert\zeta(\mathbf{r})\rvert^2\,U(t)$. This has to be done in momentum space as well, therefore it pays to choose $\zeta(\mathbf{r})$ such that it be easy to calculate its action on the state vector in momentum space. This brings us back to the restriction (5): the action of $e^{iK\xi}$ is very easy to calculate as it simply amounts to a shift by $K$ in momentum space. For $\zeta(\mathbf{r}) = e^{\pm iK\xi}$ the Hamiltonian is constant, while for $\zeta(\mathbf{r}) = \sin(K\xi)$, $\cos(K\xi)$, it is proportional to $\mp\cos(2K\xi)/2$, respectively, after dropping the constant term. This gives

$$H_{\text{I}}(t) = \mp\frac{\eta_{\text{eff}}}{2}\,U^{-1}(t)\,\cos(2K\xi)\,U(t) = \mp\frac{\eta_{\text{eff}}}{4}\,\left(e^{-4K\omega_{\text{rec}}(K-k)}e^{2iK\xi} + e^{-4K\omega_{\text{rec}}(K+k)}e^{-2iK\xi}\right). \quad (7)$$

It becomes clear how huge we gain by using interaction picture in this case. The Hamiltonian is time dependent now, but the oscillation frequency grows only linearly with $k$ instead of the quadratic growth mentioned above.

### C. Interaction elements

The functionality of these classes is summarised in Tab. III.

### 1. Particle(Orthogonal/Along)Cavity

These classes implement the interaction Hamiltonians between a cavity mode and a particle moving in 1D, either in a direction orthogonal to the cavity axis, or the direction along it, respectively.

Hence, `ParticleOrthogonalToCavity` implements

$$H = \mathrm{sign}(U_0)\sqrt{U_0\,\eta_{\mathrm{eff}}}\left(\zeta(\xi)\,a^\dagger + \zeta^*(\xi)\,a\right), \tag{8a}$$

which describes atomic stimulated absorption of a photon from the atomic pump and stimulated reemission into the cavity mode or vice versa. `ParticleAlongCavity` implements

$$H = U_0\,|f(\xi)|^2\,a^\dagger a + \mathrm{sign}(U_0)\sqrt{U_0\,\eta_{\mathrm{eff}}}\left(f^*(\xi)\,a^\dagger + f(\xi)\,a\right), \tag{8b}$$

where the first term describes atomic stimulated absorption from the cavity mode followed by stimulated reemission into the same mode.

These Hamiltonians are also implemented in interaction picture. Note that the first Hamiltonian is formally identical to the second term of the second Hamiltonian. Therefore it pays to implement this term already in a higher level in the hierarchy, so that both of these classes have access to it. This is done by the class `MovingParticleCavity` which, as we see in Fig. 4 is a parent class of both.

`ParticleAlongCavity` is either instantiated with a `MovingParticle` and an explicitly supplied parameter `etaeff`, or with a `PumpedMovingParticle`, in which case the `etaeff` parameter is taken from this latter class. The first case describes the situation when the particle pump is aligned orthogonally to the cavity axis, while the second case when it is along the axis, so that the particle, which is also moving along the axis, feels the pump potential as well.

The virtual function `FreesAdjust` defined in `Interaction` is implemented so that the cavity frequency is shifted by the interaction with the particle. In the orthogonal case this is fairly straightforward: the shift $\Delta_{\mathrm{C}} \to \Delta_{\mathrm{C}} - U_0$ is applied. In fact, the user has the choice whether it should be applied or not, in the latter case $\Delta_{\mathrm{C}}$ stands for the *shifted* frequency. With `ParticleAlongCavity`, the situation is somewhat more involved because the shift depends on the cavity mode function: for $f(\xi) = e^{\pm iK\xi}$ the shift has to be done by $U_0$, while in the $f(\xi) = \sin(K\xi)$, $\cos(K\xi)$ case by $U_0/2$ since in this case the first term of the Hamiltonian (8b) reads $U_0/2\,(1 \mp \cos(2K\xi))\,a^\dagger a$.

### 2. `ParticleCavity2D`

This class is an `Interaction` between three subsystems, and implements the Hamiltonian

$$H = U_0 \left|f(\xi_1)\right|^2 a^\dagger a + \mathrm{sign}(U_0)\sqrt{U_0\,\eta_{\mathrm{eff}}}\left(f^*(\xi_1)\,\zeta(\xi_2)\,a^\dagger + f(\xi_1)\,\zeta^*(\xi_2)\,a\right), \tag{9}$$

which describes the situation when the pumped particle is moving in two dimensions. One of the dimensions is taken care of by a `MovingParticle` class and the other one by a `PumpedMovingParticle` class — as mentioned above these classes implement *one single* spatial degree of freedom each.

### 3. `ParticleTwoModes`

It is easy to see that if we have several cavity modes then instead of (3) we have

$$\sigma \propto \sum_i g_i(\mathbf{r})\,a_i + \eta_{\mathrm{t}}(\mathbf{r}). \tag{10}$$

In the effective Hamiltonian (4b) this creates terms like

$$H \propto f^*(\xi_1)\,f(\xi_2)\,a_1^\dagger a_2 + \mathrm{h.c.}, \tag{11}$$

which describes atomic stimulated absorption of a photon from one mode and stimulated reemission into the other mode.

This cannot be described with the classes we have so far, so we need one more class `ParticleTwoModes` to cover this case as well. This closes our set of classes needed to build composite systems of an arbitrary number of (pumped) moving particles and (pumped) lossy cavity modes of different spatial configurations complying with the model (4).

`ParticleTwoModes` is an interaction between *four* subsystems, but the two spatial degrees of freedom can be the same. This describes the case of a linear cavity sustaining two modes and one particle moving along it.

### 4. `IdenticalParticles`

An interesting feature of our framework is that if we have several identical particles, it is very easy to switch between their being considered as bosons or fermions, or even distinguishable particles. All we have to do is to prepare the initial condition with the appropriate symmetry with respect to the swapping of two particles. This symmetry is then conserved during evolution.

If we consider our particles as indistinguishable, we might want to perform calculations in some occupation-number basis. This is facilitated by the `IdenticalParticles` class, which is an `Interaction` between several identical particles, which are therefore described by one single object of class `(Pumped)MovingParticle`. At its construction, an `IdenticalParticles` takes a reference to such a particle object, the number of particles, and a set of single-particle state vectors. It then constructs the occupation number basis and `Display` is implemented such that the complex amplitudes in this basis are calculated and communicated towards the user. Of course this makes sense only if the single-particle state vectors are pairwise orthogonal.

Eg for two particles and two state vectors $|\phi_1\rangle$ and $|\phi_2\rangle$ the occupation-number basis for bosons looks like

$$|2,0\rangle = |\phi_1\rangle \otimes |\phi_1\rangle\,, \tag{12a}$$

$$|1,1\rangle = \frac{1}{\sqrt{2}}\left(|\phi_1\rangle \otimes |\phi_2\rangle + |\phi_2\rangle \otimes |\phi_1\rangle\right), \tag{12b}$$

$$|0,2\rangle = |\phi_2\rangle \otimes |\phi_2\rangle\,, \tag{12c}$$

and `Display` then displays the complex amplitudes $\langle 2,0\,|\,\Psi\rangle$, $\langle 1,1\,|\,\Psi\rangle$, and $\langle 0,2\,|\,\Psi\rangle$.

In the case of indistinguishable particles it makes no sense to calculate the quantum averages for each of them separately because due to the symmetry all will be equal. Therefore, `IdenticalParticles` implements `FreesAdjust` such that the `Display` of the particles is switched off, and taken over by `IdenticalParticles`.

### D. Desideratum

We note that the above description of `IdenticalParticles` reflects the "ideal state" of the class, which allows it to be used completely generally. Clearly, for several particles and single-particle states the implementation of this involves an amount of combinatorics, and has not yet been done. Instead, in the first release of the framework `IdenticalParticles` is an interaction between *two* atoms, and calculates $\langle n_1 n_2\rangle$, where $n_1$ is the number of particles at $x < 0$ and $n_2$ at $x > 0$. Why this is useful in some cases is explained in [3]. Of course, this restriction of `IdenticalParticles` does not mean that the framework can not be used to simulate as many particles as wanted.

Atomic spontaneous emission is not implemented. In the above discussed model, where the atomic internal dynamics is eliminated, the implementation of this is rather involved, eg the jump operators have to be implemented by the interaction classes since they contain both operators $x$

and $a$ [1]. A physical problem with the spontaneous emission is that in the far detuned regime its rate is given by $\Gamma_0 = \gamma\, g^2/\Delta_{\mathrm{A}}^2$, which is much smaller than the other frequencies of the system. It therefore adds a new, very slow relaxation time scale to the system, which makes the simulations very long, practically unmanageable.

It is interesting to note that when implementing spontaneous emission, class `IdenticalParticles` gains physical significance: it has to ensure that the particle jump operators do not modify the state vector's symmetry with respect to particle exchange.

The next step in the development will be the addition of the two-level atom to the framework. This entails a number of new interaction elements, eg the term $i\left(\eta_{\mathrm{t}}^*(\mathbf{r})\sigma - \eta_{\mathrm{t}}(\mathbf{r})\sigma^\dagger\right)$ of Hamiltonian (1a) will be an `Interaction` between a two-level atom and one or several spatial degrees of freedom (`MovingParticle`s).

## VI. TEST RUNS

Testing is difficult in our case because the behaviour of the system we aim to simulate, that is, the coupled open quantum dynamics of several particles and lossy cavity-field modes is largely unknown, and constitutes an extremely rich area of active physical research — the framework is intended as a tool for this research.

Of course, utilities like `HS_Vector`, `Evoled`, `Randomized`, and maybe even `Trajectory` can be tested separately. Free elements should not present too much problem either. Interactions are, however, more problematic.

Our principle for testing interaction elements was to find parameter regimes where the action of one subsystem on the other(s) is very strong, but the back-action is negligible.

As an example, imagine a very massive pumped particle moving quickly in a direction orthogonal to a cavity. The particle is initially prepared as a very well localised wave packet. The pump is weak, so that the atom does not feel any potential, but the coupling to the cavity mode is strong, although not strong enough to create a big field that would act back on the atom. In this case the cavity field is weak, but is very sensitive to the position of the atom, on the other hand, the atom does not feel the field at all. If the particle is quick enough, it can travel several pump wavelengths before its wave packet spreads noticeably. The cavity decay rate $\kappa$ is big enough so that the field follows adiabatically even this quick atomic motion. In this case in the initial phase of the dynamics the cavity field is almost a classical field scattered by an almost classical point-like

particle. This field we can calculate explicitly:

$$\langle a \rangle = \frac{\mathrm{sign}(U_0)\sqrt{U_0\,\eta_{\mathrm{eff}}}}{\Delta_{\mathrm{C}} - U_0 + i\kappa}\zeta(x), \tag{13}$$

where $\zeta$ is the pump mode function, and $x$ is the position of the atom. An example for such a test run is displayed in Fig. 7.

**Acknowledgement**

A. V. acknowledges support from the National Scientific Fund of Hungary (Contract Nos. T043079, T049234, NF68736).

## Appendix A: DESCRIPTION OF THE MCWF METHOD

The MCWF method [6, 15, 16, 17] aims at the simulation of open quantum systems based on a stochastic ("Monte Carlo") state vector. In terms of dimensionality, this is certainly a huge advantage as compared to solving the Master equation directly. On the other hand, stochasticity requires us to run many trajectories, but the method provides an optimal sampling of the ensemble density operator so that the relative error is inversely proportional to the number of trajectories.

The optimal sampling is achieved by evolving the state vector in two steps, one deterministic and one stochastic (quantum jump). Suppose that the Master equation of the system is of the form

$$\dot{\rho} = \frac{i}{\hbar}\left[\rho, H\right] + \mathcal{L}\rho \equiv \frac{i}{\hbar}\left[\rho, H\right] + \sum_m \left( J_m \rho J_m^\dagger - \frac{1}{2}\left[J_m^\dagger J_m, \rho\right]_+ \right), \tag{A1}$$

the usual form in quantum optics. At time $t$ the system is in a state with normalised state vector $|\Psi(t)\rangle$. To obtain the state vector at time $t + \delta t$ up to first order in $\delta t$:

1. The state vector is evolved according to the non-unitary dynamics

$$i\hbar \frac{d\,|\Psi\rangle}{dt} = H_{\mathrm{nH}}\,|\Psi\rangle \tag{A2}$$

with the non-Hermitian Hamiltonian

$$H_{\mathrm{nH}} = H - \frac{i\hbar}{2}\sum_m J_m^\dagger J_m \tag{A3}$$

to obtain (up to first order in $\delta t$)

$$|\Psi_{\mathrm{nH}}(t+\delta t)\rangle = \left(1 - \frac{iH_{\mathrm{nH}}\,\delta t}{\hbar}\right)|\Psi(t)\rangle\,. \tag{A4}$$

Since $H_{\rm nH}$ is non-Hermitian, this new state vector is not normalised. The square of its norm reads

$$\langle \Psi_{\rm nH}(t+\delta t)\,|\,\Psi_{\rm nH}(t+\delta t)\rangle = \langle\Psi(t)|\left(1+\frac{iH_{\rm nH}^\dagger\,\delta t}{\hbar}\right)\left(1-\frac{iH_{\rm nH}\,\delta t}{\hbar}\right)|\Psi(t)\rangle \equiv 1-\delta p, \quad \text{(A5)}$$

where $\delta p$ reads

$$\delta p = \delta t\,\frac{i}{\hbar}\,\langle\Psi(t)|\,H_{\rm nH}-H_{\rm nH}^\dagger\,|\Psi(t)\rangle \equiv \sum_m \delta p_m, \quad \text{(A6a)}$$

$$\delta p_m = \delta t\;\langle\Psi(t)|\,J_m^\dagger J_m\,|\Psi(t)\rangle \geq 0. \quad \text{(A6b)}$$

Note that the time step $\delta t$ should be small enough so that this first-order calculation be valid. In particular, we require that

$$\delta p \ll 1. \quad \text{(A7)}$$

2. A possible quantum jump with total probability $\delta p$. For the physical interpretation of such a jump see eg Refs. [6, 17]. We choose a random number $\epsilon$ between 0 and 1, and if $\delta p < \epsilon$, which should mostly be the case, no jump occurs and for the new normalised state vector at $t+\delta t$ we take

$$|\Psi(t+\delta t)\rangle = \frac{|\Psi_{\rm nH}(t+\delta t)\rangle}{\sqrt{1-\delta p}}. \quad \text{(A8)}$$

If $\epsilon < \delta p$, on the other hand, a quantum jump occurs, and the new normalised state vector is chosen from among the different state vectors $J_m\,|\Psi(t)\rangle$ according to the probability distribution $\Pi_m = \delta p_m/\delta p$:

$$|\Psi(t+\delta t)\rangle = \sqrt{\delta t}\frac{J_m\,|\Psi(t)\rangle}{\sqrt{\delta p_m}}. \quad \text{(A9)}$$

Obviously, however, we can and must do much better than this. Indeed, assume that for some time no quantum jump occurs, and we perform Step 1 several times consecutively. This would be equivalent to evolving the Schrödinger equation with the most naive first order (Euler) method, which is known to be unstable and hence fail in most cases of interest. In our framework, we choose to use instead an adaptive step-size ODE routine, usually the embedded Runge-Kutta Cash-Karp algorithm [18]. In this case the time step is intrinsically bounded by a precision requirement in the ODE stepper, but also by the condition (A7), which is taken care of by our MCWF stepper. Since in the ODE we are now much better than $O(\delta t)$, the renormalisation of the state vector is performed exactly rather than to $O(\delta t)$ as in Eq. (A8).

In many situations it pays to use some sort of interaction picture, which means that instead of Eq. (A2) we strive to solve

$$i\hbar\frac{d\left|\Psi_{\mathrm{I}}\right\rangle}{dt}=U^{-1}\left(H_{\mathrm{nH}}U-i\hbar\frac{dU}{dt}\right)\left|\Psi_{\mathrm{I}}\right\rangle, \tag{A10}$$

where $\left|\Psi_{\mathrm{I}}\right\rangle = U^{-1}\left|\Psi\right\rangle$. Note that $U$ can be non-unitary. The two pictures are accorded after each time step, ie before the time step $\left|\Psi_{\mathrm{I}}(t)\right\rangle = \left|\Psi(t)\right\rangle$ and after the time step the transformation $\left|\Psi(t+\delta t)\right\rangle = U(\delta t)\left|\Psi_{\mathrm{I}}(t+\delta t)\right\rangle$ is performed. This we do on one hand for convenience and for compatibility with the case when no interaction picture is used, but on the other hand also because $U(t)$ is non-unitary and hence for $t\to\infty$ some of its elements will become very large, while others very small, possibly resulting in numerical problems. It is in fact advisable to avoid evaluating $U(t)$ with very large $t$ arguments.

## Appendix B: INTERACTING SYSTEMS — STATE VECTOR SLICES

The main objective of the development of the present framework was to allow users to compose composite systems at will from elementary systems and interactions already provided in the framework, and perform simulations for these composite systems. We can think of quantum optics: several atoms of different structure interacting with light fields or cavity modes. A concrete example is given in Sec. V.

Let us consider what we expect from an element of such a composite system. This element will be a class, containing all the necessary parameters specific to the given elementary system, and featuring eg a function which calculates the effect of the free elementary-system Hamiltonian $H_{\mathrm{at}}$ on a state vector. The Hamiltonian $H$ for a composite system of $N$ subsystems in terms of this Hamiltonian reads

$$H = H_0+\cdots+H_{\mathrm{at}}+\cdots+H_N+H^{\mathrm{interaction}}, \tag{B1}$$

The action of the elementary Hamiltonian $H_{\mathrm{at}}$ on a state vector $\left|\Psi\right\rangle$ expanded in a basis specified by some quantum numbers $\{i_n\}_{n=0\ldots N}$ can be written as

$$\left\langle\{i_n\}_{n=0\ldots N}\right|H_{\mathrm{at}}\left|\Psi\right\rangle=\sum_{j_{\mathrm{at}}}\left(H_{\mathrm{at}}^{\mathrm{elem}}\right)_{i_{\mathrm{at}},j_{\mathrm{at}}}\left\langle i_0,\ldots,j_{\mathrm{at}},\ldots,i_N\mid\Psi\right\rangle. \tag{B2}$$

Since at the time of developing the class of the given elementary system we do not know in which environment it will be embedded, we expect the very same piece of code to work independently of the environment. On the other hand, it has to know something about the environment because as

we see in Eq. (B2) the multiplication by the matrix of $H_{\rm at}^{\rm elem}$ has to be performed for all possible combinations of the "dummy" quantum numbers $\{i_n\}_{n\neq {\rm at}}$.

The state vector is ultimately stored as a one dimensional array (a CPA) no matter how complex the system is, and the quantum numbers $\{i_n\}_{n=0\ldots N}$ are mapped to a one dimensional index by the indexing function

$$I(i_0,\ldots,i_N)=\sum_{n=0}^{N} i_n \prod_{n+1}^{N} d_m, \tag{B3}$$

where $d$ denotes the dimension of the subsystem. Hence, the information needed by $H_{\rm at}$ about the environment can be condensed into the concept of *array slices*, which, in our framework is implemented by the `CPA_View` class. For a free system, a `CPA_View` class consists of an array `firstS` which contains the indices $I(i_0,\ldots,i_{\rm at}=0,\ldots,i_N)$ for all the possible combinations of the dummies $\{i_n\}_{n\neq {\rm at}}$ and an integer $\texttt{stride}=\prod_{\rm at+1}^{N} d_m$.

To each element of the array `firstS` of a `CPA_View` there corresponds a `CPA_Slice` which contains one single index `first` and the integer `stride`. One can say that `CPA_Slice` is the iterator type of `CPA_View`. The index corresponding to a subsystem quantum number $i_{\rm at}$ for a given set of the dummy quantum numbers can then be calculated from the slice alone as

$$I\left(i_{\rm at}|\,\{i_n\}_{n\neq {\rm at}}\right)=\texttt{first}+\texttt{stride}\times i_{\rm at}. \tag{B4}$$

All the environment-independent implementation of $H_{\rm at}$ and eventually that of every operator acting on a subsystem *at* of a composite system has to see from the environment is a `CPA_View`. Having received a `CPA_View` as a parameter all an elementary Hamiltonian $H_{\rm at}$ has to do is to iterate over the dummy indices condensed into `firstS` and apply the same matrix $H_{\rm at}^{\rm elem}$ on the state-vector slice specified by the corresponding `CPA_Slice`. This concept is realized by `H` and `H_elem`, cf Sec. IV B.

`CPA_View` is essentially an array of `CPA_Slice`s, we just save resources by storing `stride`, which is the characteristic of the given subsystem embedded in the given environment, only once.

As discussed in Sec. IV B, interactions are also "elements" in our framework. An interaction Hamiltonian operates on several subsystems, therefore its `CPA_View` has to contain as many `stride`s, each corresponding to the stride characteristic for the given subsystem in the given embedding environment.

The concept of array slices, the relationship between `CPA_Slice` and `CPA_View`, and the fact

that a `CPA_View` represents a way of looking on the state vector is further exposed in Fig. 8.

---


[1] A. Vukics, J. Janszky, and P. Domokos, J. Phys. B: At. Mol. Opt. Phys. **38**, 1453 (2005).

[2] A. Vukics and P. Domokos, Phys. Rev. A **72**, 031401 (2005).

[3] C. Maschler, H. Ritsch, A. Vukics, and P. Domokos (2005), quant-ph/0512101.

[4] A. Vukics, Ph.D. thesis, University of Szeged, Szeged (2006), URL `http://optics.szfki.kfki.hu/~vukics/thesispdf.pdf`.

[5] P. Domokos and H. Ritsch, J. Opt. Soc. Am. B **20**, 1098 (2003).

[6] K. Molmer, Y. Castin, and J. Dalibard, J. Opt. Soc. Am. B **10**, 524 (1993).

[7] R. Schack and T. A. Brun (1996), quant-ph/9608004.

[8] S. M. Tan, J. Opt. B: Quant. Semiclass. Opt. **1**, 424 (1999).

[9] G. Collecutt and P. D. Drummond, Comp. Phys. Commun. **142**, 219 (2001).

[10] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi, *GNU Scientific Library — Reference Manual*, GNU (2006), URL `http://www.gnu.org/software/gsl/manual/`.

[11] B. Stroustrup, *The C++ Programming Language* (Addison-Wesley, 1997), 3rd ed.

[12] N. Gisin and I. C. Percival, J. Phys. A **25**, 5677 (1992).

[13] L. Diosi, Phys. Lett. A **114**, 451 (1986).

[14] A. Vukics, P. Domokos, and H. Ritsch, J. Opt. B: Quant. Semiclass. Opt. **6**, 143 (2004).

[15] H. J. Carmichael, J. Opt. Soc. Am. B **4**, 1588 (1987).

[16] J. Dalibard, Y. Castin, and K. Molmer, Phys. Rev. Lett. **68**, 580 (1992).

[17] R. Dum, P. Zoller, and H. Ritsch, Phys. Rev. A **45**, 4879 (1992).

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C* (Cambridge, 1992), URL `http://www.nr.com/`.

```
0  #include "PumpedLossyMode.H"
1  #include "ParticleAlongCavity.H"
2  #include "ParticleTwoModes.H"
3  #include "Composite.H"
4  #include "Trajectory.H"
5
6  int main(int, char*) {
7
8    // Instantiate Frees
9    MovingParticle p(omrec,fin); // Free0
10   LossyMode mPlus(DeltaC,kappa,cutoff1); // Free1
11   PumpedLossyMode mMinus(DeltaC,kappa,eta,cutoff2); // Free2
12
13   // Instantiate Interactions
14   ParticleAlongCavity pc1(&mPlus,&p,U0,etaeff,K,Plus); // exp(iKx) mode (Plus)
15   ParticleAlongCavity pc2(&mMinus,&p,U0,etaeff,K,Minus); // exp(-iKx) mode (Minus)
16   ParticleTwoModes ptm(&pc1,&pc2);
17
18   // Instantiate Composite
19   vector<SubsystemsInteraction> i(1,SubsystemsInteraction(pc1,1,0));
20   i.push_back(SubsystemsInteraction(pc2,2,0));
21   vector<size_t> N(1,1); N.push_back(2); N.push_back(0); N.push_back(0); // N={1,2,0,0}
22   i.push_back(SubsystemsInteraction(ptm,N));
23   Composite c(i);
24
25   // Initial condition
26   HS_Vector Psi=WavePacket(p,x0,k0,xsig)*Coherent(mPlus,alpha)*Coherent(mMinus,beta);
27   // Instantiate Trajectory
28   Trajectory t(Psi,c,seed);
29   // Run Trajectory
30   RunTrajectory(t,T);
31
32 }
```

Figure 1: Full driver for one particle in a ring cavity sustaining two travelling-wave modes with opposite wave vectors, the $-K$ mode being pumped. The definition of parameters (`omrec`, `fin`, etc.) has been omitted for the sake of compactness.

```
0  // Instantiate Frees
1  LossyMode m(DeltaC,kappa,cutoff); // Free0
2  PumpedMovingParticle p(omrec,etaeff,fin,K,Sin); // Free1, Free2 --- only one instant!
3
4  // Instantiate Interactions
5  ParticleOrthogonalToCavity pc(m,p,U0); // only one instant!
6  IdenticalParticles id(p,2,Psileft,Psiright);
7
8  // Instantiate Composite
9  std::vector<SubsystemsInteraction> i(1,SubsystemsInteraction(pc,0,1));
10 i.push_back(SubsystemsInteraction(pc,0,2)); i.push_back(SubsystemsInteraction(id,1,2));
11 Composite c(i);
12
13 // Initial condition
14 HS_Vector Psi=Coherent(m,alpha)*TwoParticleState(p,SuperFluid);
```

Figure 2: The essential part of the driver for two identical pumped particles moving orthogonal to the axis of a cavity sustaining one single sinusoidal mode — or otherwise, two identical particles moving in a one dimensional optical lattice with the cavity aligned orthogonally to the lattice.

```
# MCWFS Driver Parameters:
# seed=1001
# eps=1e-05
# dplimit=0.1
# Displaying in every 10 timestep

# Composite Dissipative System of Dimension 768

# Subsystem Nr. 0
# Moving Particle:
# omrec=1
# Spatial Degree of Freedom finesse=6

# Subsystem Nr. 1
# Lossy Mode:
# Z=(1,0) kappa=1 N=3

# Subsystem Nr. 2
# Lossy Mode:
# Z=(1,0) kappa=1 N=4
# eta=(0.3,-0.07)
# Field from pump: (0.3,-0.07)

# 1 <-> 0 Interaction
# Particle Moving along Cavity
# Particle-Cavity Interaction Unot=-1 K=1 (etaeff=-1). Mode function type: Plus

# 2 <-> 0 Interaction
# Particle Moving along Cavity
# Particle-Cavity Interaction Unot=-1 K=1 (etaeff=-1). Mode function type: Minus

# 1 <-> 2  <-> 0 Interaction
# Particle Two Modes

0           0                   0.1       2.78      1.57      0.302             0.0181    0.0181    0.0999    0.0899            0.338     0.334     0.298     0.497
0.0724291   0.00773444          0.0469    2.81      1.58      0.387             0.0081    0.00811   0.0269    0.083             0.354     0.35      0.368     0.465
0.154641    0.00945583          -0.00308  2.9       1.55      0.689             0.0106    0.0106    -0.0451   0.075             0.375     0.373     0.431     0.428
0.242855    0.0087605           -0.0382   3.05      1.31      1.19              0.0256    0.0257    -0.105    0.0667            0.395     0.398     0.476     0.391
0.321498    0.00827827          -0.0504   3.19      0.979     1.52              0.046     0.0459    -0.142    0.0594            0.407     0.418     0.497     0.358
0.399775    0.00795456          -0.0446   3.35      0.65      1.7               0.0694    0.0692    -0.163    0.0519            0.412     0.431     0.502     0.326
0.469162    0.00501979          -0.0257   3.48      0.419     1.77              0.0905    0.0901    -0.168    0.0449            0.41      0.438     0.494     0.299
0.535599    0.00697113          0.00238   3.61      0.193     1.8               0.109     0.109     -0.164    0.0378            0.405     0.438     0.478     0.273
0.602433    0.00773779          0.0384    3.72      -0.0591   1.8               0.126     0.126     -0.153    0.0306            0.396     0.434     0.456     0.248
0.659532    0.00602507          0.0735    3.82      -0.194    1.79              0.138     0.138     -0.138    0.0244            0.387     0.427     0.434     0.227
0.722639    0.00553655          0.115     3.91      -0.252    1.8               0.148     0.149     -0.119    0.0176            0.376     0.417     0.409     0.205
0.782856    0.00483279          0.156     4         -0.332    1.8               0.155     0.157     -0.0992   0.0112            0.364     0.406     0.384     0.185
0.844433    0.00701458          0.198     4.07      -0.373    1.79              0.16      0.163     -0.0781   0.00506           0.352     0.394     0.361     0.165
0.903525    0.00639796          0.237     4.14      -0.314    1.79              0.162     0.167     -0.0583   -0.000578         0.342     0.382     0.34      0.148
0.962645    0.00450127          0.274     4.21      -0.118    1.81              0.162     0.168     -0.0396   -0.0059           0.332     0.371     0.322     0.131
```

Figure 3: Typical output of the ring-cavity driver of Fig. 1. The first two columns are time and time step, respectively, then, separated by tab characters, the data stemming from the different subsystems follows: columns 3-6 contain the data from subsystem Nr. 0 `MovingParticle`, columns 7-10 and 11-14 from the two cavity modes. The interaction elements make no output in this example.

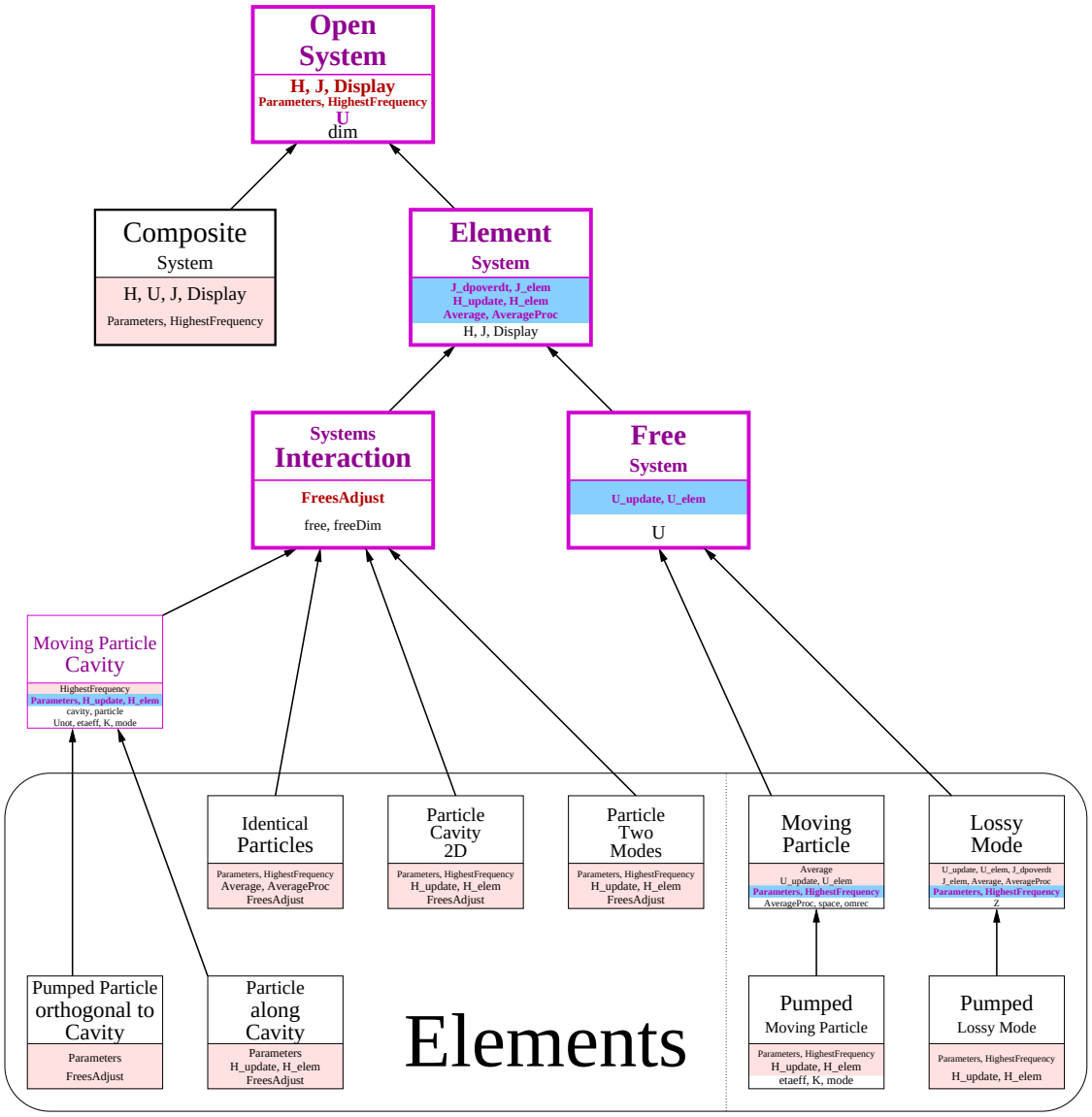


Figure 4: Class inheritance hierarchy starting from the almost purely abstract interface `OpenSystem`, the interface that all simulated systems has to provide for our `Trajectory` driver. At the bottom of the hierarchy we have provided an example set of `Element`s taken from CQED with moving particles. These may serve as building blocks for `Composite`s. The colour code: magenta-framed classes are abstract classes, black-framed ones are concrete types; arrows denote class inheritance; in each class the most important functions are displayed — purely virtual ones in red, virtual ones in magenta and concrete ones in black; the functions displayed in the salmon stripes belong to the private part of the class while the blue and white stripes refer to the protected and public part, respectively. The displayed functions are partly documented in the text, and partly in the source code.

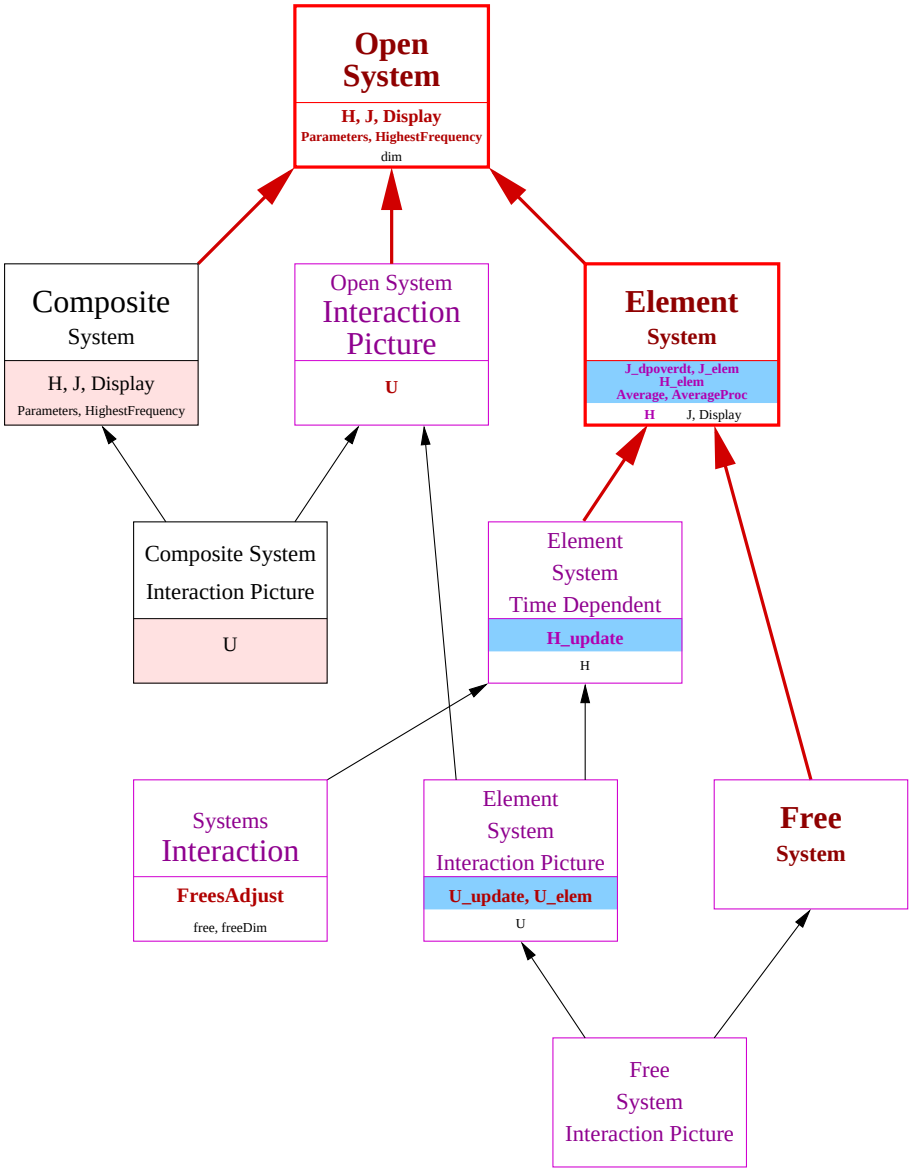


Figure 5: Alternative design featuring a completely separate branch for systems using interaction picture. Red-framed classes are virtual bases, and red arrows denote virtual inheritance.

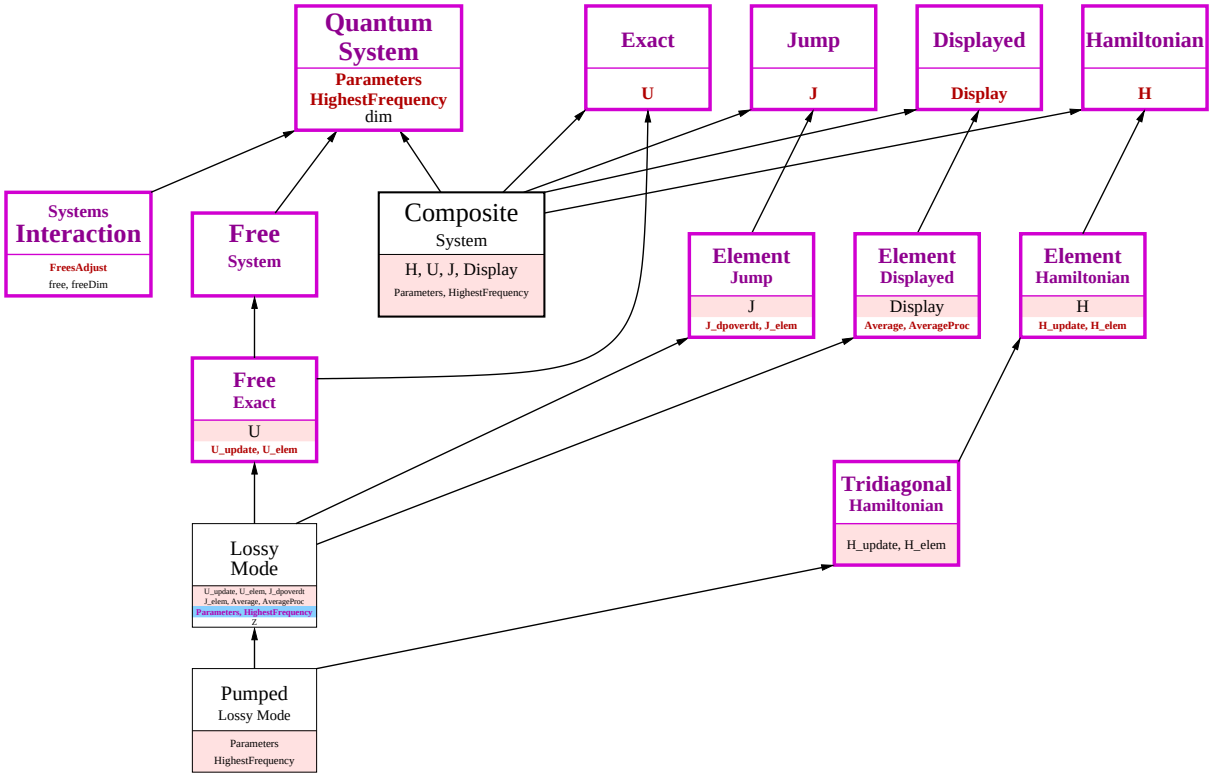


Figure 6: The design actually used in the framework. The advantage over the first design is that the fundamental functions `H`, `U`, `J`, and `Display` are declared as *pure* virtual, and therefore it is very clear which class has implemented which function. Still, it does not use virtual bases as the second design. The function of `Element` has ceased to exist so this class is omitted, we have instead a set of classes `ElementHamiltonian` etc. `Composite` then deals separately with `Frees` and `Interactions`. Logically, the root of the hierarchy is not called `OpenSystem` anymore, since the jump function is declared outside this class, but merely `QuantumSystem`. As an example we have plotted `LossyMode` and `PumpedLossyMode` to show how concrete elements fit into this hierarchy. Note that eg `H` cannot even be called for `LossyMode`, only for `PumpedLossyMode` since the first is *not* derived from the `Hamiltonian` class.

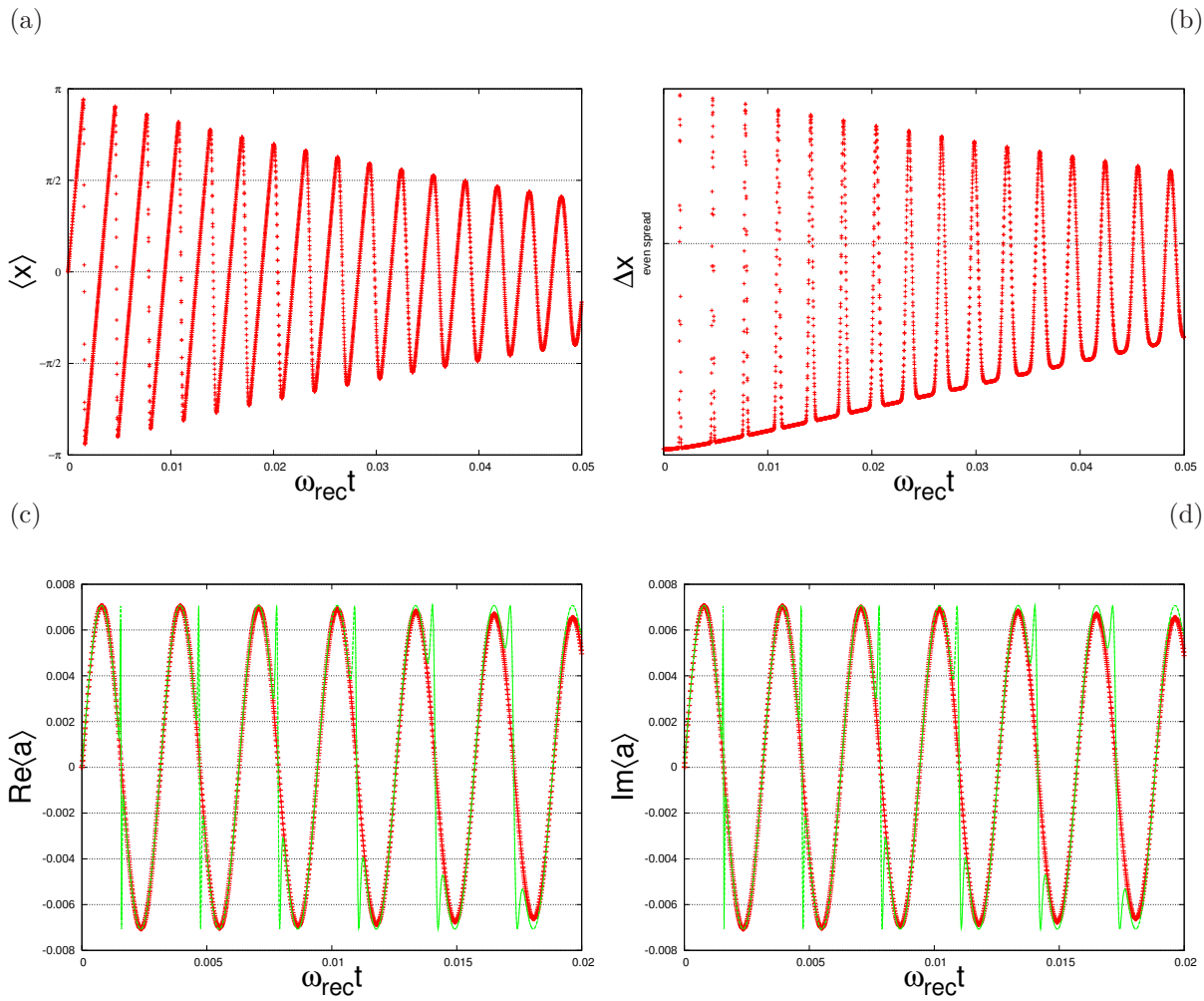


Figure 7: Massive pumped particle moving quickly in a direction orthogonal to the axis of a cavity. (a) Expectation value of the atom's position. Each time the atom goes out of the quantisation volume at $x = \pi$, it comes back in at $x = -\pi$ due to periodic boundary condition. (b) Spread of the atomic wave packet. (c) & (d) Real and imaginary part of the scattered field in the cavity, the green lines corresponding to the estimation (13).

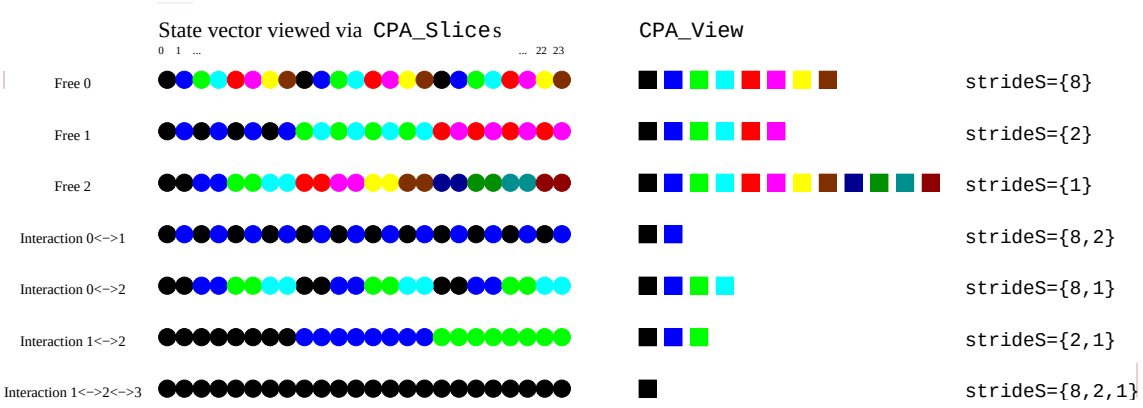


Figure 8: The state vector of a system consisting of three subsystems with dimensions 3, 4 and 2, covered by different sets of `CPA_Slice`s corresponding to the free subsystems and the interactions between the subsystems. One slice is the set of indices displayed in the same colour. A `CPA_View` is essentially an array of slices with the modification that the `strideS` are stored only once.